Autoimpute Documentation

Joseph Kearney, Shahid Barkat

May 24, 2023

User Guide

Autoimpute is a Python package for analysis and implementation of Imputation Methods.

This page introduces users to the package and documents its features.

Check out the package on github, or head to our website to walk through some tutorials.

CHAPTER 1

Getting Started

The sections below provide a high level overview of the `Autoimpute` package. This page takes you through installation, dependencies, main features, imputation methods supported, and basic usage of the package. It also provides links to get in touch with the authors, review our lisence, and review how to contribute.

1.1 Installation

- Download `Autoimpute` with `pip install autoimpute`.
- If `pip` cached an older version, try `pip install --no-cache-dir --upgrade autoimpute`.
- If you want to work with the development branch, use the script below:

Development

```
git clone -b dev --single-branch https://github.com/kearnz/autoimpute.git
cd autoimpute
python setup.py install
```

1.2 Versions and Dependencies

- Python 3.8+
- Dependencies:
  - `numpy`
  - `scipy`
  - `pandas`
  - `statsmodels`
  - `scikit-learn`

  - `xgboost`
  - `pymc`
  - `seaborn`
  - `missingno`

A note for Windows Users:

- Autoimpute 0.13.0+ has not been tested on windows and can't verify support for pymc. Historically we've had some issues with pymc on windows.
- Autoimpute works on Windows but users may have trouble with pymc for bayesian methods. (See discourse)
- Users may receive a runtime error `'can't pickle fortran objects'` when sampling using multiple chains.
- There are a couple of things to do to try to overcome this error:
  - Reinstall theano and pymc. Make sure to delete .theano cache in your home folder.
  - Upgrade joblib in the process, which is reponsible for generating the error (pymc uses joblib under the hood).
  - Set `cores=1` in `pm.sample`. This should be a last resort, as it means posterior sampling will use 1 core only. Not using multiprocessing will slow down bayesian imputation methods significantly.
- Reach out and let us know if you've worked through this issue successfully on Windows and have a better solution!

1.3 Main Features

- Utility functions and basic visualizations to explore missingness patterns
- Missingness classifier and automatic missing data test set generator
- Native handling for categorical variables (as predictors and targets of imputation)
- Single and multiple imputation classes for `pandas DataFrames`
- Analysis methods and pooled parameter inference using multiply imputed datasets
- Numerous imputation methods, as specified in the table below:

1.4 Imputation Methods Supported

Univariate	Multivariate	Time Series / Interpolation
Mean	Linear Regression	Linear
Median	Binomial Logistic Regression	Quadratic
Mode	Multinomial Logistic Regression	Cubic
Random	Stochastic Regression	Polynomial
Norm	Bayesian Linear Regression	Spline
Categorical	Bayesian Binary Logistic Regression	Time-weighted
	Predictive Mean Matching	Next Obs Carried Backward
	Local Residual Draws	Last Obs Carried Forward

1.5 Example Usage

Autoimpute is designed to be user friendly and flexible. When performing imputation, Autoimpute fits directly into `scikit-learn` machine learning projects. Imputers inherit from sklearn's `BaseEstimator` and `TransformerMixin` and implement `fit` and `transform` methods, making them valid Transformers in an sklearn pipeline.

Right now, there are two `Imputer` classes we'll work with:

```
from autoimpute.imputations import SingleImputer, MultipleImputer, MiceImputer
si = SingleImputer() # pass through data once
mi = MultipleImputer() # pass through data multiple times
mice = MiceImputer() # pass through data multiple times and iteratively optimize␣
↪imputations in each column
```

Imputations can be as simple as:

```
# simple example using default instance of MiceImputer
imp = MiceImputer()

# fit transform returns a generator by default, calculating each imputation method␣
↪lazily
imp.fit_transform(data)
```

Or quite complex, such as:

```
# create a complex instance of the MiceImputer
# Here, we specify strategies by column and predictors for each column
# We also specify what additional arguments any `pmm` strategies should take
imp = MiceImputer(
    n=10,
    strategy={"salary": "pmm", "gender": "bayesian binary logistic", "age": "norm"},
    predictors={"salary": "all", "gender": ["salary", "education", "weight"]},
    imp_kwgs={"pmm": {"fill_value": "random"}},
    visit="left-to-right",
    return_list=True
)

# Because we set return_list=True, imputations are done all at once, not evaluated␣
↪lazily.
# This will return M*N, where M is the number of imputations and N is the size of␣
↪original dataframe.
imp.fit_transform(data)
```

Autoimpute also extends supervised machine learning methods from `scikit-learn` and `statsmodels` to apply them to multiply imputed datasets (using the `MiceImputer` under the hood). Right now, Autoimpute supports linear regression and binary logistic regression. Additional supervised methods are currently under development.

As with Imputers, Autoimpute's analysis methods can be simple or complex:

```
from autoimpute.analysis import MiLinearRegression

# By default, use statsmodels OLS and MiceImputer()
simple_lm = MiLinearRegression()

# fit the model on each multiply imputed dataset and pool parameters
simple_lm.fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
# get summary of fit, which includes pooled parameters under Rubin's rules
# also provides diagnostics related to analysis after multiple imputation
simple_lm.summary()

# make predictions on a new dataset using pooled parameters
predictions = simple_lm.predict(X_test)

# Control both the regression used and the MiceImputer itself
multiple_imputer_arguments = dict(
    n=3,
    strategy={"salary": "pmm", "gender": "bayesian binary logistic", "age": "norm"},
    predictors={"salary": "all", "gender": ["salary", "education", "weight"]},
    imp_kwgs={"pmm": {"fill_value": "random"}},
    scaler=StandardScaler(),
    visit="left-to-right",
    verbose=True
)
complex_lm = MiLinearRegression(
    model_lib="sklearn", # use sklearn linear regression
    mi_kwgs=multiple_imputer_arguments # control the multiple imputer
)

# fit the model on each multiply imputed dataset
complex_lm.fit(X_train, y_train)

# get summary of fit, which includes pooled parameters under Rubin's rules
# also provides diagnostics related to analysis after multiple imputation
complex_lm.summary()

# make predictions on new dataset using pooled parameters
predictions = complex_lm.predict(X_test)
```

Note that we can also pass a pre-specified `MiceImputer` to either analysis model instead of using `mi_kwgs`. The option is ours, and it's a matter of preference. If we pass a pre-specified `MiceImputer`, anything in `mi_kwgs` is ignored, although the `mi_kwgs` argument is still validated.

```
from autoimpute.imputations import MiceImputer
from autoimpute.analysis import MiLinearRegression

# create a multiple imputer first
custom_imputer = MiceImputer(n=3, strategy="pmm", return_list=True)

# pass the imputer to a linear regression model
complex_lm = MiLinearRegression(mi=custom_imputer, model_lib="statsmodels")

# proceed the same as the previous examples
complex_lm.fit(X_train, y_train).predict(X_test)
complex_lm.summary()
```

For a deeper understanding of how the package works and its features, see our tutorials website.

1.6 Creators and Maintainers

- Joseph Kearney – @kearnz

- Shahid Barkat - @shabarka

See the Authors page to get in touch!

1.7 License

Distributed under the MIT license. See LICENSE for more information.

1.8 Contributing

Guidelines for contributing to our project. See CONTRIBUTING for more information.

1.9 Contributor Code of Conduct

Adapted from Contributor Covenant, version 1.0.0. See Code of Conduct for more information.

CHAPTER 2

Utility Methods

Methods to numerically assess patterns in missing data.

This module is a collection of methods to explore missing data and its patterns. The module's methods are heavily influenced by those found in section 4.1 of Flexible Imputation of Missing Data (Van Buuren). Their main purpose is to identify trends and patterns in missing data that can help inform what type of imputation method may apply or what cautions to take when performing imputations in general.

autoimpute.utils.patterns.**flux**(*data*)

Caclulates inbound, influx, outbound, outflux, pobs, for DataFrame.

Port of Van Buuren's flux method in R. Calculates: - *pobs*: Proportion observed (column from the *proportions* method). - *ainb*: Average inbound statistic. - aout: Average outbound statistic. - influx: Influx coefficient, Ij (from the *influx* method). - outflux: Outflux coefficient, Oj (from the *outflux* method).

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate relevant statistics.

Returns

one column for each summary statistic. Columns of DataFrame equal the name of the summary statistics. Indices of DataFrame equal the original DataFrame columns.

Return type pd.DataFrame

autoimpute.utils.patterns.**get_stat_for**(*func*, *data*)

Generic method to get a missing data statistic from data.

This method can be used directly with helper methods, but this behavior is discouraged. Instead, use specific public methods below. These special methods utilize this function internally to compute summary statistics.

Parameters

- **func** (*function*) – Function that calculates a statistic.
- **data** (*pd.DataFrame*) – DataFrame on which to run the function.

Returns Output from statistic chosen.

Return type np.ndarray

autoimpute.utils.patterns.**inbound**(*data*)
Calculates proportion of usable cases (Ijk) from Van Buuren 4.1.

Method ported from VB, called "inbound statistic", Ijk. Ijk = 1 if variable Yk observed in all records where Yj missing. Used to quickly select potential predictors Yk for imputing Yj. High values are preferred.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate inbound statistic.

Returns

inbound statistic between each of the features. Inbound between a feature and itself is 0.

Return type pd.DataFrame

autoimpute.utils.patterns.**influx**(*data*)
Calculates the influx coefficient (Ij) from Van Buuren 4.1.

Method ported from VB, called "influx coefficient", Ij. Ij = # pairs (Yj,Yk) w/ Yj missing & Yk observed / # observed data cells. Value depends on the proportion of missing data of the variable. Influx of a completely observed variable is equal to 0. Influx for completely missing variables is equal to 1. For two variables with the same proportion of missing data: - Variable with higher influx is better connected to the observed data. - Variable with higher influx might thus be easier to impute.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate influx coefficient.

Returns influx coefficient for each column.

Return type pd.DataFrame

autoimpute.utils.patterns.**md_locations**(*data*, *both=False*)
Produces locations where values are missing in a DataFrame.

Takes in a DataFrame and identifies locations where data is complete or missing. Normally, fully complete issues warning, and fully incomplete throws error, but this method simply shows missingness locations, so the general standard for mixed complete-missing pattern not necessary. Method marks 1 = missing, 0 = not missing.

Parameters

- **data** (*pd.DataFrame*) – DataFrame to find missing & complete observations.
- **both** (*boolean, optional*) – return data along with missingness indicator. Defaults to False, so just missingness indicator returned.

Returns

missingness indicator DataFrame OR pd.DataFrame: missingness indicator DataFrame concatenated column-wise

with original DataFame.

Return type pd.DataFrame

Raises TypeError – if data is not a DataFrame. Error raised through decorator.

autoimpute.utils.patterns.**md_pairs**(*data*)
Calculates pairwise missing data statistics.

This method mimics the behavior of MICE md.pairs. - rr: response-response pairs - rm: response-missing pairs - mr: missing-response pairs - mm: missing-missing pairs Returns a square matrix for each, where n = number of columns.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate pairwise stats.

Returns keys are pair types, values are DataFrames w/ pair stats.

Return type dict

Raises TypeError – if data is not a DataFrame. Error raised through decorator.

autoimpute.utils.patterns.**md_pattern**(*data*)

Calculates row-wise missing data statistics in input data.

Method is a port of md.pattern method from VB 4.1. The number of rows indicates the number of different row patterns of missingness. The 'nmis' column is the number of missing values in a given row pattern. The 'count' is number of total rows with a given row pattern. In this method, 0 = missing, 1 = missing.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate missing data pattern.

Returns

DataFrame with missing data pattern and two additional columns w/ row-wise stats: *count* and *nmis*.

Return type pd.DataFrame

autoimpute.utils.patterns.**nullility_corr**(*data*, *method='pearson'*)

Calculates the nullility correlation between features in a DataFrame.

Leverages pandas method to calculate correlation of nullility. Note that this method drops NA values to compute correlation. It also employs *check_missingness* decorator to ensure DataFrame not fully missing. If a DataFrame is fully observed, nothing is returned, as there is no nullility.

Parameters

- **data** (*pd.DataFrame*) – DataFrame to calculate nullility correlation.
- **method** (*string, optional*) – correlation method to use. Default pearson, but spearman should be used with categorical or ordinal encoding.

Returns DataFrame with nullility correlation b/w each feature.

Return type pd.DataFrame

Raises

- TypeError – If data not pd.DataFrame. Raised through decorator.
- ValueError – If DataFrame values all missing and none complete. Also raised through decorator.
- ValueError – If method for correlation not an accepted method.

autoimpute.utils.patterns.**nullility_cov**(*data*)

Calculates the nullility covariance between features in a DataFrame.

Leverages pandas method to calculate covariance of nullility. Note that this method drops NA values to compute covariance. It also employs *check_missingness* decorator to ensure DataFrame not fully missing. If a DataFrame is fully observed, nothing is returned, as there is no nullility.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate nullility covariance.

Returns DataFrame with nullility covariance b/w each feature.

Return type pd.DataFrame

Raises

- TypeError – If data not pd.DataFrame. Raised through decorator.
- ValueError – If DataFrame values all missing and none complete. Also raised through decorator.

autoimpute.utils.patterns.**outbound**(*data*)
Calculates the outbound statistic (Ojk) from Van Buuren 4.1.

Method ported from VB, called "outbound statistic", Ojk. Ojk measures how observed data Yj connect to rest of missing data. Ojk = 1 if Yj observed in all records where Yk is missing. Used to evaluate whether Yj is a potential predictor for imputing Yk. High values are preferred.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate outbound statistic.

Returns

outbound statistic between each of the features. Outbound between a feature and itself is 0.

Return type pd.DataFrame

autoimpute.utils.patterns.**outflux**(*data*)
Calculates the outflux coefficient (Oj) from Van Buuren 4.1.

Method ported from VB, called "outflux coefficient", Oj. Oj = # pairs w/ Yj observed and Yk missing / # incomplete data cells. Value depends on the proportion of missing data of the variable. Outflux of a completely observed variable is equal to 1. Outflux of a completely missing variable is equal to 0. For two variables having the same proportion of missing data: - Variable with higher outflux is better connected to the missing data. - Variable with higher outflux more useful for imputing other variables.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate outflux coefficient.

Returns outflux coefficient for each column.

Return type pd.DataFrame

autoimpute.utils.patterns.**proportions**(*data*)
Calculates the proportions of the data missing and data observed.

Method calculates two arrays: - *poms*: Proportion of missing data. - *pobs*: Proportion of observed data.

Parameters **data** (*pd.DataFrame*) – DataFrame to calculate proportions.

Returns

two columns, one for *poms* and one for *pobs*. The sum of each row should equal 1. Index = original data cols.

Return type pd.DataFrame

Raises TypeError – if data not DataFrame. Error raised through decorator.

CHAPTER 3

Deletion and Imputation Strategies

This section documents deletion and imputation strategies within `Autoimpute`.

Deletion is implemented through a single function, `listwise_delete`, documented below.

Imputation strategies are implemented as classes. The authors of this package refer to these classes as "series-imputers". Each series-imputer maps to an imputation method - either univariate or multivariate - that imputes missing values within a pandas Series. The imputation methods are the workhorses of the DataFrame Imputers, the `SingleImputer`, `MultipleImputer`, and `MiceImputer`. Refer to the *imputers documentation* for more information on the DataFrame Imputers.

For more information regarding the relationship between DataFrame Imputers and series-imputers, refer to the following tutorial. The tutorial covers series-imputers in detail as well as the design patterns behind `AutoImpute` Imputers.

3.1 Deletion Methods

`autoimpute.imputations.`**`listwise_delete`**(*data*, *inplace=False*, *verbose=False*)

Delete all rows from a DataFrame where any missing values exist.

Deletion is one way to handle missing values. This method removes any records that have a missing value in any of the features. This package focuses on imputation, not deletion. That being said, listwise deletion is a necessary component of any imputation package, as its the default method most people (and software) use to handle missing data.

Parameters

- **`data`** (*`pd.DataFrame`*) – DataFrame used to delete missing rows.
- **`inplace`** (*`boolean, optional`*) – perform operation inplace. Defaults to False.
- **`verbose`** (*`boolean, optional`*) – print information to console. Defaults to False.

Returns rows with missing values removed.

Return type pd.DataFrame

Raises `ValueError` – columns with all data missing. Raised through decorator.

3.2 Imputation Strategies

Manage the series imputations folder from the autoimpute package.

This module handles imports from the series imputations folder that should be accessible whenever someone imports autoimpute.imputations.series. Although these imputers are stand-alone classes, their direct use is discouraged. More robust imputers from the dataframe folder delegate work to these imputers whenever their respective strategies are requested.

This module handles *from autoimpute.imputations.series import* * with the __all__ variable below. This command imports the main public classes and methods from autoimpute.imputations.series.

class autoimpute.imputations.series.**DefaultUnivarImputer**(*num_imputer=<class 'autoimpute.imputations.series.mean.MeanImputer'>*, *cat_imputer=<class 'autoimpute.imputations.series.mode.ModeImputer'>*, *num_kwgs=None*, *cat_kwgs={'fill_strategy': 'random'}*)

Impute missing data using default methods for univariate imputation.

This imputer is the default for univariate imputation. The imputer determines how to impute based on the column type of each column in a dataframe. The imputer can be used directly, but such behavior is discouraged. DefaultUnivarImputer does not have the flexibility / robustness of more complex imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="default univariate").

__init__(*num_imputer=<class 'autoimpute.imputations.series.mean.MeanImputer'>*, *cat_imputer=<class 'autoimpute.imputations.series.mode.ModeImputer'>*, *num_kwgs=None*, *cat_kwgs={'fill_strategy': 'random'}*)

Create an instance of the DefaultUnivarImputer class.

The dataframe imputers delegate work to the DefaultUnivarImputer if strategy="default univariate" The DefaultUnivarImputer then determines how to impute numerical and categorical columns by default. It does so by passing its arguments to the DefaultBaseImputer, which handles validation and instantiation of numerical and categorical imputers.

Parameters

- **num_imputer** (*Imputer, Optional*) – valid Imputer for numerical data. Default is MeanImputer.
- **cat_imputer** (*Imputer, Optional*) – valid Imputer for categorical data. Default is ModeImputer.
- **num_kwgs** (*dict, optional*) – Keyword args for numerical imputer. Default is None.
- **cat_kwgs** (*dict, optional*) – keyword args for categorical imputer. Default is {"fill_strategy": "random"}.

Returns self. Instance of class.

fit(*X*, *y=None*)

Defer fit to the DefaultBaseImputer.

impute(*X*)
Defer transform to the DefaultBaseImputer.

class autoimpute.imputations.series.**DefaultTimeSeriesImputer**(*num_imputer=<class 'autoimpute.imputations.series.interpolation.Interpo cat_imputer=<class 'autoimpute.imputations.series.mode.ModeImputer' num_kwgs={'fill_strategy': 'linear'}, cat_kwgs={'fill_strategy': 'random'}*)

Impute missing data using default methods for time series.

This imputer is the default imputer for time series imputation. The imputer determines how to impute based on the column type of each column in a dataframe. The imputer can be used directly, but such behavior is discouraged. DefaultTimeSeriesImputer does not have the flexibility / robustness of more complex imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="default time").

__init__(*num_imputer=<class 'autoimpute.imputations.series.interpolation.InterpolateImputer'>, cat_imputer=<class 'autoimpute.imputations.series.mode.ModeImputer'>, num_kwgs={'fill_strategy': 'linear'}, cat_kwgs={'fill_strategy': 'random'}*)
Create an instance of the DefaultTimeSeriesImputer class.

The dataframe imputers delegate work to the DefaultTimeSeriesImputer if strategy="default time". The DefaultTimeSeriesImputer then determines how to impute numerical and categorical columns by default. It does so by passing its arguments to the DefaultBaseImputer, which handles validation and instantiation of default numerical and categorical imputers.

Parameters

- **num_imputer** (*Imputer, Optional*) – valid Imputer for numerical data. Default is InterpolateImputer.
- **cat_imputer** (*Imputer, Optional*) – valid Imputer for categorical data. Default is ModeImputer.
- **num_kwgs** (*dict, optional*) – Keyword args for numerical imputer. Default is {"strategy": "linear"}.
- **cat_kwgs** (*dict, optional*) – keyword args for categorical imputer. Default is {"fill_strategy": "random"}.

Returns self. Instance of class.

fit(*X*, *y=None*)
Defer fit to the DefaultBaseImputer.

impute(*X*)
Defer transform to the DefaultBaseImputer.

class autoimpute.imputations.series.**DefaultPredictiveImputer**(*num_imputer=<class 'autoimpute.imputations.series.pmm.PMMImputer' cat_imputer=<class 'autoimpute.imputations.series.logistic_regression.M num_kwgs=None, cat_kwgs=None*)

Impute missing data using default methods for prediction.

This imputer is the default imputer for the MultipleImputer class. When an end-user does not supply a strategy, the DefaultPredictiveImputer determines how to impute based on the column type of each column in a dataframe. The imputer can be used directly, but such behavior is discouraged. DefaultPredictiveImputer does not have the flexibility / robustness of more complex imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="default predictive").

__init__(*num_imputer=<class 'autoimpute.imputations.series.pmm.PMMImputer'>*, *cat_imputer=<class 'autoimpute.imputations.series.logistic_regression.MultinomialLogisticImputer'>*, *num_kwgs=None*, *cat_kwgs=None*)

Create an instance of the DefaultPredictiveImputer class.

The dataframe imputers delegate work to DefaultPredictiveImputer if strategy="default predictive" or no strategy given when class is instantiated. The DefaultPredictiveImputer determines how to impute numerical and categorical columns by default. It does so by passing its arguments to the DefaultBaseImputer, which handles validation and instantiation of default numerical and categorical imputers.

Parameters

- **num_imputer** (*Imputer, Optional*) – valid Imputer for numerical data. Default is PMMImputer.
- **cat_imputer** (*Imputer, Optional*) – valid Imputer for categorical data. Default is MultiLogisticImputer.
- **num_kwgs** (*dict, optional*) – Keyword args for numerical imputer. Default is None.
- **cat_kwgs** (*dict, optional*) – keyword args for categorical imputer. Default is None.

Returns self. Instance of class.

fit(*X*, *y*)

Defer fit to the DefaultBaseImputer.

impute(*X*)

Defer transform to the DefaultBaseImputer.

class autoimpute.imputations.series.**MeanImputer**

Impute missing values with the mean of the observed data.

This imputer imputes missing values with the mean of observed data. The imputer can be used directly, but such behavior is discouraged. MeanImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="mean").

__init__()

Create an instance of the MeanImputer class.

fit(*X*, *y*)

Fit the Imputer to the dataset and calculate the mean.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)

Convenience method to perform fit and imputation in one go.

impute(*X*)

Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Missing values in a given dataset are replaced with the respective mean from fit.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns float – imputed dataset.

class autoimpute.imputations.series.**MedianImputer**

Impute missing values with the median of the observed data.

This imputer imputes missing values with the median of observed data. The imputer can be used directly, but such behavior is discouraged. MedianImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="median").

__init__()

Create an instance of the MedianImputer class.

fit(*X*, *y=None*)

Fit the Imputer to the dataset and calculate the median.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)

Convenience method to perform fit and imputation in one go.

impute(*X*)

Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Missing values in a given dataset are replaced with the respective median from fit.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns float – imputed dataset.

class autoimpute.imputations.series.**ModeImputer**(*fill_strategy=None*)

Impute missing values with the mode of the observed data.

The mode imputer calculates the mode of the observed dataset and uses it to impute missing observations. In the case where there are more than one mode, the user can supply a *fill_strategy* to choose the mode. The imputer can be used directly, but such behavior is discouraged. ModeImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="mode").

__init__(*fill_strategy=None*)

Create an instance of the ModeImputer class.

Parameters **fill_strategy** (*str, Optional*) – strategy to pick mode, if multiple. Default is None, which means first mode taken. Options include None, first, last, random. First, None -> select first of modes. Last -> select the last of modes. Random -> randomly sample from modes with replacement.

fill_strategy

Property getter to return the value of fill_strategy property.

fit(*X*, *y=None*)

Fit the Imputer to the dataset and calculate the mode.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute(*X*)
Perform imputations using the statistics generated from fit.

This method handles the actual imputation. Missing values in a given dataset are replaced with the mode observed from fit. Note that there can be more than one mode. If more than one mode, use the fill_strategy to determine how to use the modes.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns float or np.array – imputed dataset.

class autoimpute.imputations.series.**RandomImputer**
Impute missing data using random draws from observed data.

The RandomImputer samples with replacement from observed data. The imputer can be used directly, but such behavior is discouraged. RandomImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="random").

__init__()
Create an instance of the RandomImputer class.

fit(*X*, *y=None*)
Fit the Imputer to the dataset and get unique observed to sample.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute(*X*)
Perform imputations using the statistics generated from fit.

The transform method handles the actual imputation. Each missing value in a given dataset is replaced with a random draw from unique set of observed values determined during the fit stage.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset

class autoimpute.imputations.series.**NormImputer**
Impute missing data with draws from normal distribution.

The NormImputer constructs a normal distribution using the sample mean and variance of the observed data. The imputer then randomly samples from this distribution to impute missing data. The imputer can be used directly, but such behavior is discouraged. NormImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="norm").

__init__()
Create an instance of the NormImputer class.

fit (*X*, *y=None*)
Fit Imputer to dataset and calculate mean and sample variance.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute (*X*, *y*)
Convenience method to perform fit and imputation in one go.

impute (*X*)
Perform imputations using the statistics generated from fit.

The transform method handles the actual imputation. It constructs a normal distribution using the sample mean and variance from fit. It then imputes missing values with a random draw from the respective distribution.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset.

class autoimpute.imputations.series.**CategoricalImputer**
Impute missing data w/ draw from dataset's categorical distribution.

The categorical imputer computes the proportion of observed values for each category within a discrete dataset. The imputer then samples the distribution to impute missing values with a respective random draw. The imputer can be used directly, but such behavior is discouraged. CategoricalImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="categorical").

__init__ ()
Create an instance of the CategoricalImputer class.

fit (*X*, *y=None*)
Fit the Imputer to the dataset and calculate proportions.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute (*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute (*X*)
Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Transform constructs a categorical distribution for each feature using the proportions of observed values from fit. It then imputes missing values with a random draw from the respective distribution.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset.

class autoimpute.imputations.series.**NOCBImputer** (*end=None*)
Impute missing data by carrying the next observation backward.

NOCBImputer carries the next observation backward to impute missing data. The imputer can be used directly, but such behavior is discouraged. NOCBImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="nocb").

__init__(*end=None*)
Create an instance of the NOCBImputer class.

Parameters **end** (*any, optional*) – can be any value to impute end if end is missing. Default is None, which ends up taking last observed value found. Can also use "mean" to end with mean of the series.

Returns self. Instance of class.

fit(*X*, *y=None*)
Fit the Imputer to the dataset and calculate the mean.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute(*X*)
Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Missing values in a given dataset are replaced with the next observation carried backward.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset.

class autoimpute.imputations.series.**LOCFImputer**(*start=None*)
Impute missing values by carrying the last observation forward.

LOCFImputer carries the last observation forward to impute missing data. The imputer can be used directly, but such behavior is discouraged. LOCFImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="locf").

__init__(*start=None*)
Create an instance of the LOCFImputer class.

Parameters **start** (*any, optional*) – can be any value to impute first if first is missing. Default is None, which ends up taking first observed value found. Can also use "mean" to start with mean of the series.

Returns self. Instance of class.

fit(*X*, *y=None*)
Fit the Imputer to the dataset.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute(*X*)
Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Missing values in a given dataset are replaced with the last observation carried forward.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset.

class autoimpute.imputations.series.**InterpolateImputer**(*fill_strategy='linear'*, *start=None*, *end=None*, *order=None*)
Impute missing values using interpolation techniques.

The InterpolateImputer imputes missing values uses a valid pd.Series interpolation strategy. See __init__ method docs for supported strategies. The imputer can be used directly, but such behavior is discouraged. InterpolateImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="interpolate").

__init__(*fill_strategy='linear'*, *start=None*, *end=None*, *order=None*)
Create an instance of the InterpolateImputer class.

Parameters

- **fill_strategy** (*str, Optional*) – type of interpolation to perform Default is linear. Other strategies supported include: *time*, *quadratic*, *cubic*, *spline*, *barycentric*, *polynomial*.
- **start** (*int, Optional*) – value to impute if first number in Series is missing. Default is None, but first valid used when required for quadratic, cubic, polynomial.
- **end** (*int, Optional*) – value to impute if last number in Series is missing. Default is None, but last valid used when required for quadratic, cubic, polynomial.
- **order** (*int, Optional*) – if strategy is spline or polynomial, order must be number. Otherwise not considered.

Returns self. Instance of the class.

fill_strategy
Property getter to return the value of fill_strategy property.

fit(*X*, *y=None*)
Fit the Imputer to the dataset. Nothing to calculate.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute(*X*)
Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Missing values in a given dataset are replaced with results from interpolation.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset.

class autoimpute.imputations.series.**LeastSquaresImputer**(***kwargs*)

Impute missing values using predictions from least squares regression.

The LeastSquaresImputer produces predictions using the least squares methodology. The prediction from the line of best fit given a set of predictors become the imputations. To implement least squares, the imputer wraps the sklearn LinearRegression class. The imputer can be used directly, but such behavior is discouraged. LeastSquaresImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="least squares").

__init__(***kwargs*)

Create an instance of the LeastSquaresImputer class.

Parameters ****kwargs** – keyword arguments passed to LinearRegression

fit(*X*, *y*)

Fit the Imputer to the dataset by fitting linear model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute(*X*, *y*)

Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute(*X*)

Generate imputations using predictions from the fit linear model.

The impute method returns the values for imputation. Missing values in a given dataset are replaced with the predictions from the least squares regression line of best fit. This transform method returns those predictions.

Parameters **X** (*pd.DataFrame*) – predictors to determine imputed values.

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**StochasticImputer**(***kwargs*)

Impute missing values adding error to least squares regression preds.

The StochasticImputer predicts using the least squares methodology. The imputer then samples from the regression's error distribution and adds the random draw to the prediction. This draw adds the stochastic element to the imputations. The imputer can be used directly, but such behavior is discouraged. StochasticImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="stochastic").

__init__(***kwargs*)

Create an instance of the StochasticImputer class.

Parameters ****kwargs** – keyword arguments passed to LinearRegression.

fit(*X*, *y*)
Fit the Imputer to the dataset by fitting linear model.

The fit step also generates predictions on the observed data. These predictions are necessary to derive the mean_squared_error, which is passed as a parameter to the impute phase. The MSE is used to create the normal error distribution from which the imptuer draws.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute(*X*, *y*)
Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute

Returns imputated dataset.

Return type np.array

impute(*X*)
Generate imputations using predictions from the fit linear model.

The impute method returns the values for imputation. Missing values in a given dataset are replaced with the predictions from the least squares regression line of best fit plus a random draw from the normal error distribution.

Parameters **X** (*pd.DataFrame*) – predictors to determine imputed values.

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**BinaryLogisticImputer**(***kwargs*)
Impute missing values w/ predictions from binary logistic regression.

The BinaryLogisticImputer produces predictions using logsitic regression with two classes. The class predictions given a set of predictors become the imputations. To implement logistic regression, the imputer wraps the sklearn LogisticRegression class with a default solver (liblinear). The imputer can be used directly, but such behavior is discouraged. BinaryLogisticImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="binary logistic").

__init__(***kwargs*)
Create an instance of the BinaryLogisticImputer class.

Parameters ****kwargs** – keyword arguments passed to LogisticRegresion.

fit(*X*, *y*)
Fit the Imputer to the dataset by fitting logistic model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute(*X*, *y*)
Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute(*X*)
Generate imputations using predictions from the fit logistic model.

The impute method returns the values for imputation. Missing values in a given dataset are replaced with the predictions from the logistic regression class specification.

Parameters **X** (*pd.DataFrame*) – predictors to determine imputed values.

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**MultinomialLogisticImputer**(***kwargs*)
Impute missing values w/ preds from multinomial logistic regression.

The MultinomialLogisticImputer produces predictions w/ logsitic regression with more than two classes. Class predictions given a set of predictors become the imputations. To implement logistic regression, the imputer wraps the sklearn LogisticRegression class with a default solver (saga) and default *multi_class* set to multinomial. The imputer can be used directly, but such behavior is discouraged. MultinomialLogisticImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="multinomial logistic").

__init__(***kwargs*)
Create an instance of the MultiLogisticImputer class.

Parameters ****kwargs** – keyword arguments passed to LogisticRegression.

fit(*X*, *y*)
Fit the Imputer to the dataset by fitting logistic model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute(*X*, *y*)
Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute(*X*)
Generate imputations using predictions from the fit logistic model.

The impute method returns the values for imputation. Missing values in a given dataset are replaced with the predictions from the logistic regression class specification.

Parameters **X** (*pd.DataFrame*) – predictors to determine imputed values.

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**BayesianLeastSquaresImputer**(***kwargs*)

Impute missing values using bayesian least squares regression.

The BayesianLeastSquaresImputer produces predictions using the bayesian approach to least squares. Prior distributions are fit for the model parameters of interest (alpha, beta, epsilon). Imputations for missing values are samples from posterior predictive distribution of each missing point. To implement bayesian least squares, the imputer utlilizes the pymc library. The imputer can be used directly, but such behavior is discouraged. BayesianLeastSquaresImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="bayesian least squares").

__init__(***kwargs*)

Create an instance of the BayesianLeastSquaresImputer class.

The class requires multiple arguments necessary to create priors for a bayesian linear regression equation. The regression is: alpha + beta * X + epsilson. Because paramaters are treated as random variables, we must specify their distributions, including the parameters of those distributions. In thie init method we also include arguments used to sample the posterior distributions.

Parameters

- ****kwargs** – default keyword arguments used for bayesian analysis. Note - kwargs popped for default arguments defined below. Rest of kwargs passed as params to sampling (see pymc).
- **am** (*float, Optional*) – mean of alpha prior. Default 0.
- **asd** (*float, Optional*) – std. deviation of alpha prior. Default 10.
- **bm** (*float, Optional*) – mean of beta priors. Default 0.
- **bsd** (*float, Optional*) – std. deviation of beta priors. Default 10.
- **sig** (*float, Optional*) – parameter of sigma prior. Default 1.
- **sample** (*int, Optional*) – number of posterior samples per chain. Default = 1000. More samples, longer to run, but better approximation of the posterior & chance of convergence.
- **tune** (*int, Optional*) – parameter for tuning. Draws done in addition to sample. Default = 1000.
- **init** (*str, Optional*) – MCMC algo to use for posterior sampling. Default = 'auto'. See pymc docs for more info on choices.
- **fill_value** (*str, Optional*) – How to draw from the posterior to create imputations. Default is None. 'random' and 'mean' supported for explicit options.

fit(*X*, *y*)

Fit the Imputer to the dataset by fitting bayesian model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute(*X*, *y*)
Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute(*X*, *k=None*)
Generate imputations using predictions from the fit bayesian model.

The transform method returns the values for imputation. Missing values in a given dataset are replaced with the samples from the posterior predictive distribution of each missing data point.

Parameters

- **X** (*pd.DataFrame*) – predictors to determine imputed values.
- **k** (*integer*) – optional, pass if and only if receiving from MICE

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**BayesianBinaryLogisticImputer**(***kwargs*)
Impute missing values using bayesian binary losgistic regression.

The BayesianBinaryLogisticImputer produces predictions using the bayesian approach to logistic regression. Prior distributions are fit for the model parameters of interest (alpha, beta, epsilon). Imputations for missing values are samples from the posterior predictive distribution of each missing point. To implement bayesian logistic regression, the imputer uses the pymc library. The imputer can be used directly, but such behavior is discouraged. BayesianBinaryLogisticImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="bayesian binary logistic").

__init__(***kwargs*)
Create an instance of the BayesianBinaryLogisticImputer class.

The class requires multiple arguments necessary to create priors for a bayesian logistic regression equation. The parameters are the same as linear regression, but the regression equation is transformed using pymc's invlogit method. Because paramaters are treated as random variables, we must specify their distributions, including the parameters of those distributions. In the init method we also include arguments used to sample the posterior distributions.

Parameters

- ****kwargs** – default keyword arguments used for bayesian analysis. Note - kwargs popped for default arguments defined below. Rest of kwargs passed as params to sampling (see pymc).
- **am** (*float, Optional*) – mean of alpha prior. Default 0.
- **asd** (*float, Optional*) – std. deviation of alpha prior. Default 10.
- **bm** (*float, Optional*) – mean of beta priors. Default 0.
- **bsd** (*float, Optional*) – std. deviation of beta priors. Default 10.

- **thresh** (*float, Optional*) – threshold for class membership. Default 0.5. Max = 1, min = 0. Tune threshhold depending on class imbalance. Same as with logistic regression equation.
- **sample** (*int, Optional*) – number of posterior samples per chain. Default = 1000. More samples, longer to run, but better approximation of the posterior & chance of convergence.
- **tune** (*int, Optional*) – parameter for tuning. Draws done in addition to sample. Default = 1000.
- **init** (*str, Optional*) – MCMC algo to use for posterior sampling. Default = 'auto'. See pymc docs for more info on choices.
- **fill_value** (*str, Optional*) – How to draw from the posterior to create imputations. Default is None. 'random' and 'mean' supported for explicit options.

fit (*X*, *y*)

Fit the Imputer to the dataset by fitting bayesian model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute (*X*, *y*)

Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute (*X*, *k=None*)

Generate imputations using predictions from the fit bayesian model.

The impute method returns the values for imputation. Missing values in a given dataset are replaced with the samples from the posterior predictive distribution of each missing data point.

Parameters

- **X** (*pd.DataFrame*) – predictors to determine imputed values.
- **k** (*integer*) – optional, pass if and only if receiving from MICE

Returns imputated dataset.

Return type np.array

class autoimpute.imputations.series.**PMMImputer** (***kwargs*)

Impute missing values using predictive mean matching.

The PMMIMputer produces predictions using a combination of bayesian approach to least squares and least squares itself. For each missing value PMM finds the *n* closest neighbors from a least squares regression prediction set, and samples from the corresponding true values for y of each of those *n* predictions. The imputation is the resulting sample. To implement bayesian least squares, the imputer utlilizes the pymc library. The imputer can be used directly, but such behavior is discouraged. PmmImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="pmm").

__init__(***kwargs*)

Create an instance of the PMMImputer class.

The class requires multiple arguments necessary to create priors for a bayesian linear regression equation and least squares itself. Therefore, PMM arguments include all of those seen in bayesian least squares and least squares itself. New parameters include *neighbors*, or the number of neighbors that PMM uses to sample observed.

Parameters

- ****kwargs** – default keyword arguments for lm & bayesian analysis. Note - kwargs popped for default arguments defined below. Next set of kwargs popped and sent to linear regression. Rest of kwargs passed as params to sampling (see pymc).
- **am** (*float, Optional*) – mean of alpha prior. Default 0.
- **asd** (*float, Optional*) – std. deviation of alpha prior. Default 10.
- **bm** (*float, Optional*) – mean of beta priors. Default 0.
- **bsd** (*float, Optional*) – std. deviation of beta priors. Default 10.
- **sig** (*float, Optional*) – parameter of sigma prior. Default 1.
- **sample** (*int, Optional*) – number of posterior samples per chain. Default = 1000. More samples, longer to run, but better approximation of the posterior & chance of convergence.
- **tune** (*int, Optional*) – parameter for tuning. Draws done in addition to sample. Default = 1000.
- **init** (*str, Optional*) – MCMC algo to use for posterior sampling. Default = 'auto'. See pymc docs for more info on choices.
- **fill_value** (*str, Optional*) – How to draw from the posterior to create imputations. Default is "random". 'random' and 'mean' supported for explicit options.
- **neighbors** (*int, Optional*) – number of neighbors. Default is 5. Value should be greater than 0 and less than # observed, although anything greater than 10-20 generally too high unless dataset is massive.
- **fit_intercept** (*bool, Optional*) – sklearn LinearRegression param.
- **normalize** (*bool, Optional*) – sklearn LinearRegression param.
- **copy_x** (*bool, Optional*) – sklearn LinearRegression param.
- **n_jobs** (*int, Optional*) – sklearn LinearRegression param.

fit(*X*, *y*)

Fit the Imputer to the dataset by fitting bayesian and LS model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute(*X*, *y*)

Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.

- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute(*X*)

Generate imputations using predictions from the fit bayesian model.

The transform method returns the values for imputation. Missing values in a given dataset are replaced with the random selection from the PMM process. Again, PMM imputes actually observed values, and the observed values are selected by finding the closest least squares predictions to a given prediction from the bayesian model.

Parameters **X** (*pd.DataFrame*) – predictors to determine imputed values.

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**LRDImputer**(***kwargs*)

Impute missing values using local residual draws.

The LRDImputer produces predictions using a combination of bayesian approach to least squares and least squares itself. For each missing value LRD finds the *n* closest neighbors from a least squares regression prediction set, and samples from the corresponding true values for y of each of those *n* predictions. The imputation is the resulting sample plus the residual, or the distance between the prediction and the neighbor. To implement bayesian least squares, the imputer utlilizes the pymc library. The imputer can be used directly, but such behavior is discouraged. LRDImputer does not have the flexibility / robustness of dataframe imputers, nor is its behavior identical. Preferred use is MultipleImputer(strategy="lrd").

__init__(***kwargs*)

Create an instance of the LRDImputer class.

The class requires multiple arguments necessary to create priors for a bayesian linear regression equation and least squares itself. Therefore, LRD arguments include all of those seen in bayesian least squares and least squares itself. New parameters include *neighbors*, or the number of neighbors that LRD uses to sample observed.

Parameters

- ****kwargs** – default keyword arguments for lm & bayesian analysis. Note - kwargs popped for default arguments defined below. Next set of kwargs popped and sent to linear regression. Rest of kwargs passed as params to sampling (see pymc).
- **am** (*float, Optional*) – mean of alpha prior. Default 0.
- **asd** (*float, Optional*) – std. deviation of alpha prior. Default 10.
- **bm** (*float, Optional*) – mean of beta priors. Default 0.
- **bsd** (*float, Optional*) – std. deviation of beta priors. Default 10.
- **sig** (*float, Optional*) – parameter of sigma prior. Default 1.
- **sample** (*int, Optional*) – number of posterior samples per chain. Default = 1000. More samples, longer to run, but better approximation of the posterior & chance of convergence.
- **tune** (*int, Optional*) – parameter for tuning. Draws done in addition to sample. Default = 1000.
- **init** (*str, Optional*) – MCMC algo to use for posterior sampling. Default = 'auto'. See pymc docs for more info on choices.

- **fill_value** (*str, Optional*) – How to draw from the posterior to create imputations. Default is "random". 'random' and 'mean' supported for explicit options.
- **neighbors** (*int, Optional*) – number of neighbors. Default is 5. Value should be greater than 0 and less than # observed, although anything greater than 10-20 generally too high unless dataset is massive.
- **fit_intercept** (*bool, Optional*) – sklearn LinearRegression param.
- **copy_x** (*bool, Optional*) – sklearn LinearRegression param.
- **n_jobs** (*int, Optional*) – sklearn LinearRegression param.

fit (*X*, *y*)
Fit the Imputer to the dataset by fitting bayesian and LS model.

Parameters

- **X** (*pd.Dataframe*) – dataset to fit the imputer.
- **y** (*pd.Series*) – response, which is eventually imputed.

Returns self. Instance of the class.

fit_impute (*X*, *y*)
Fit impute method to generate imputations where y is missing.

Parameters

- **X** (*pd.Dataframe*) – predictors in the dataset.
- **y** (*pd.Series*) – response w/ missing values to impute.

Returns imputed dataset.

Return type np.array

impute (*X*)
Generate imputations using predictions from the fit bayesian model.

The transform method returns the values for imputation. Missing values in a given dataset are replaced with the random selection from the LRD process. Again, LRD imputes actually observed values, and the observed values are selected by finding the closest least squares predictions to a given prediction from the bayesian model.

Parameters **X** (*pd.DataFrame*) – predictors to determine imputed values.

Returns imputed dataset.

Return type np.array

class autoimpute.imputations.series.**NormUnitVarianceImputer**
Impute missing values assuming normally distributed data with unknown mean and *known* variance.

__init__ ()
Create an instance of the NormUnitVarianceImputer class.

fit (*X*, *y*)
Fit the Imputer to the dataset and calculate the mean.

Parameters

- **X** (*pd.Series*) – Dataset to fit the imputer.
- **y** (*None*) – ignored, None to meet requirements of base class

Returns self. Instance of the class.

fit_impute(*X*, *y=None*)
Convenience method to perform fit and imputation in one go.

impute(*X*)
Perform imputations using the statistics generated from fit.

The impute method handles the actual imputation. Missing values in a given dataset are replaced with the respective mean from fit.

Parameters **X** (*pd.Series*) – Dataset to impute missing data from fit.

Returns np.array – imputed dataset.

CHAPTER 4

DataFrame Imputers

This section documents the DataFrame Imputers within `Autoimpute`.

DataFrame Imputers are the primary feature of the package. The `SingleImputer` imputes each column within a DataFrame one time, while the `MultipleImputer` imputes each column within a DataFrame multiple times using independent runs. Under the hood, the `MultipleImputer` actually creates separate instances of the `SingleImputer` to handle each run. The `MiceImputer` takes the `MultipleImputer` one step futher, iteratively improving imputations in each column `k` times for each `n` runs the `MultipleImputer` performs.

The base class for all imputers is the `BaseImputer`. While you should not use the `BaseImputer` directly unless you're creating your own imputer class, you should understand what it provides the other imputers. The `BaseImputer` also contains the strategy "key-value store", or the methods that `autoimpute` currently supports.

4.1 Base Imputer

class autoimpute.imputations.**BaseImputer**(*strategy*, *imp_kwgs*, *visit*)

Building blocks for more advanced DataFrame imputers.

The BaseImputer is not a stand-alone class and thus serves no purpose other than as a parent to Imputers. Therefore, the BaseImputer should not be used directly unless creating an Imputer. That being said, all DataFrame Imputers should inherit from BaseImputer. It contains base functionality for any new DataFrame Imputer, and it holds the set of strategies that make up this imputation library.

univariate_strategies

univariate imputation methods. | Key = imputation name; Value = function to perform imputation. | *univariate default* mean for numerical, mode for categorical. | *time default* interpolate for numerical, mode for categorical. | *mean* imputes missing values with the average of the series. | *median* imputes missing values with the median of the series. | *mode* imputes missing values with the mode of the series. | Method handles more than one mode (see ModeImputer for info). | *random* imputes random choice from set of series unique vals. | *norm* imputes series w/ random draws from normal distribution. | Mean and std calculated from observed values of the series. | *categorical* imputes series using random draws from pmf. | Proportions calculated from non-missing category instances. | *interpolate* imputes series using chosen interpolation method. | Default is linear. See InterpolateImputer for more info. | *locf* imputes

series carrying last observation moving forward. | *nocb* imputes series carrying next observation moving backward. | *normal unit variance* imputes using unit variance w/ norm.

Type dict

predictive_strategies
predictive imputation methods. | Key = imputation name; Value = function to perform imputation. | *predictive default* pmm for numerical,logistic for categorical. | *least squares* predict missing values from linear regression. | *binary logistic* predict missing values with 2 classes. | *multinomial logistic* predict missing values with multiclass. | *stochastic* linear regression+random draw from norm w/ mse std. | *bayesian least squares* draw from the posterior predictive | distribution for each missing value, using OLS model. | *bayesian binary logistic* draw from the posterior predictive | distribution for each missing value, using logistic model. | *pmm* imputes series using predictive mean matching. PMM is a | semi-supervised method using bayesian & hot-deck imputation. | *lrd* imputes series using local residual draws. LRD is a | semi-supervised method using bayesian & hot-deck imputation.

Type dict

__init__ (*strategy*, *imp_kwgs*, *visit*)
Initialize the BaseImputer.

Parameters

- **strategy** (*str, iter, dict; optional*) – strategies for imputation. Default value is str -> *predictive default*. If str, single strategy broadcast to all series in DataFrame. If iter, must provide 1 strategy per column. Each method w/in iterator applies to column with same index value in DataFrame. If dict, must provide key = column name, value = imputer. Dict the most flexible and PREFERRED way to create custom imputation strategies if not using the default. Dict does not require method for every column; just those specified as keys.
- **imp_kwgs** (*dict, optional*) – keyword arguments for each imputer. Default is None, which means default imputer created to match specific strategy. imp_kwgs keys can be either columns or strategies. If strategies, each column given that strategy is instantiated with same arguments.
- **visit** (*str, None*) – order to visit columns for imputation. Default is *default*, which implements *left-to-right*. More strategies (random, monotone, etc.) TBD.

__weakref__
list of weak references to the object (if defined)

imp_kwgs
Property getter to return the value of imp_kwgs.

strategy
Property getter to return the value of the strategy property.

visit
Property getter to return the value of the visit property.

4.2 Single Imputer

class autoimpute.imputations.**SingleImputer**(*strategy='default predictive'*, *predictors='all'*, *imp_kwgs=None*, *copy=True*, *seed=None*, *visit='default'*)
Techniques to impute Series with missing values one time.

The SingleImputer class takes a DataFrame and performs imputations on each Series within the DataFrame. The Imputer does one pass for each column, and it supports numerous imputation methods for each column.

The SingleImputer delegates imputation to respective SeriesImputers, each of which maps to a specific strategy supported by the SingleImputer. Most of the SeriesImputers are inductive (fit and transform for new data). Transductive SeriesImputers (such as InterpolateImputer) still perform a "mock" fit stage but do all the imputation work in the transform step. The fit stage is performed to remain consistent with the sklearn API. The class is a valid sklearn transformer that can be used in an sklearn Pipeline because it inherits from the TransformerMixin and implements both fit and transform methods.

__init__ (*strategy='default predictive'*, *predictors='all'*, *imp_kwgs=None*, *copy=True*, *seed=None*, *visit='default'*)

Create an instance of the SingleImputer class.

As with sklearn classes, all arguments take default values. Therefore, SingleImputer() creates a valid class instance. The instance is used to set up a SingleImputer and perform checks on arguments.

Parameters

- **strategy** (*str, iter, dict; optional*) – strategy for single imputer. Default value is str –> *predictive default*. See BaseImputer for all available strategies. If str, single strategy broadcast to all series in DataFrame. If iter, must provide 1 strategy per column. Each method w/in iterator applies to column with same index value in DataFrame. If dict, must provide key = column name, value = imputer. Dict the most flexible and PREFERRED way to create custom imputation strategies if not using the default. Dict does not require method for every column; just those specified as keys.
- **predictors** (*str, iter, dict, optional*) – defaults to *all*, i.e. use all predictors. If *all*, every column will be used for every class prediction. If a list, subset of columns used for all predictions. If a dict, specify which columns to use as predictors for each imputation. Columns not specified in dict but present in *strategy* receive *all* other cols as preds. Note predictors are IGNORED for univariate imputation methods, so specifying is meaningless unless strategy is predictive.
- **imp_kwgs** (*dict, optional*) – keyword args for each SeriesImputer. Default is None, which means default imputer created to match specific strategy. *imp_kwgs* keys can be either columns or strategies. If strategies, each column given that strategy is instantiated with same arguments. When strategy is *default*, *imp_kwgs* is ignored.
- **copy** (*bool, optional*) – create copy of DataFrame or operate inplace. Default value is True. Copy created.
- **seed** (*int, optional*) – seed setting for reproducible results. Defualt is None. No validation, but values should be integer.

fit (*X*, *y=None*, *imp_ixs=None*)

Fit specified imputation methods to each column within a DataFrame.

The fit method calculates the *statistics* necessary to later transform a dataset (i.e. perform actual imputations). Inductive methods calculate statistic on the fit data, then impute new missing data with that value. Most currently supported methods are inductive.

It's important to note that we have to fit X regardless of whether any data is missing. Transform step may have missing data if new data is used, so fit each column that appears in the given strategies.

Parameters

- **X** (*pd.DataFrame*) – pandas DataFrame on which imputer is fit.
- **y** (*pd.Series, pd.DataFrame Optional*) – response. Default is None. Determined interally in fit method. Arg is present to remain compatible with sklearn Pipelines.

- **imp_ixs** (*dict*) – Dictionary of lists of indices that indicate which data elements to impute per column or None to identify from missing elements per column

Returns instance of the SingleImputer class.

Return type self

Raises

- ValueError – error in specification of strategies. Raised through *check_strategy_fit*. See its docstrings for more info.
- ValueError – error in specification of predictors. Raised through *check_predictors_fit*. See its docstrings for more info.

fit_transform(*X*, *y=None*, ***trans_kwargs*)

Convenience method to fit then transform the same dataset.

Parameters

- **X** (*pd.DataFrame*) – DataFrame used for fit and transform steps.
- **y** (*pd.DataFrame, pd.Series, Optional*) – response. Default is None. Set internally by *fit* method.
- ****trans_kwargs** – dict, optional args for bayesian.

Returns imputed in place or copy of original.

Return type X (pd.DataFrame)

transform(*X*, *imp_ixs=None*, ***trans_kwargs*)

Impute each column within a DataFrame using fit imputation methods.

The transform step performs the actual imputations. Given a dataset previously fit, *transform* imputes each column with it's respective imputed values from fit (in the case of inductive) or performs new fit and transform in one sweep (in the case of transductive).

Parameters

- **X** (*pd.DataFrame*) – DataFrame to impute (same as fit or new data).
- **imp_ixs** (*dict*) – Dictionary of lists of indices that indicate which data elements to impute per column or None to identify from missing elements per column
- ****trans_kwargs** – dict, optional args for bayesian.

Returns imputed in place or copy of original.

Return type X (pd.DataFrame)

Raises ValueError – same columns must appear in fit and transform. Raised through _transform_strategy_validator.

4.3 Multiple Imputer

class autoimpute.imputations.**MultipleImputer**(*n=5*, *strategy='default predictive'*, *predictors='all'*, *imp_kwgs=None*, *seed=None*, *visit='default'*, *return_list=False*)

Techniques to impute Series with missing values multiple times.

The MultipleImputer class applies imputation multiple times. It leverages the methods found in the BaseImputer. This imputer passes all the work for each imputation to the SingleImputer, but it controls the arguments each imputer receives. The args are flexible depending on what the user specifies for each imputation.

Note that the Imputer allows for one imputation method per column only. Therefore, the behavior of *strategy* is the same as the SingleImputer, but the predictors are allowed to change for each imputation.

__init__(*n=5*, *strategy='default predictive'*, *predictors='all'*, *imp_kwgs=None*, *seed=None*, *visit='default'*, *return_list=False*)

Create an instance of the MultipleImputer class.

As with sklearn classes, all arguments take default values. Therefore, MultipleImputer() creates a valid class instance. The instance is used to set up an imputer and perform checks on arguments.

Parameters

- **n** (*int, optional*) – number of imputations to perform. Default is 5. Value must be greater than or equal to 1.
- **strategy** (*str, iter, dict; optional*) – strategy for single imputer. Default value is str –> *predictive default*. See BaseImputer for all available strategies. If str, single strategy broadcast to all series in DataFrame. If iter, must provide 1 strategy per column. Each method w/in iterator applies to column with same index value in DataFrame. If dict, must provide key = column name, value = imputer. Dict the most flexible and PREFERRED way to create custom imputation strategies if not using the default. Dict does not require method for every column; just those specified as keys.
- **predictors** (*str, iter, dict, optional*) – defaults to all, i.e. use all predictors. If all, every column will be used for every class prediction. If a list, subset of columns used for all predictions. If a dict, specify which columns to use as predictors for each imputation. Columns not specified in dict but present in *strategy* receive *all* other cols as preds.
- **imp_kwgs** (*dict, optional*) – keyword arguments for each imputer. Default is None, which means default imputer created to match specific strategy. imp_kwgs keys can be either columns or strategies. If strategies, each column given that strategy is instantiated with same arguments. When strategy is *default*, imp_kwgs is ignored.
- **seed** (*int, optional*) – seed setting for reproducible results. Defualt is None. No validation, but values should be integer.
- **return_list** (*bool, optional*) – return m as list or generator. Default is False. m imputations returned as generator. More memory efficient. return as list if return_list=True

fit(*X*, *y=None*)

Fit imputation methods to each column within a DataFrame.

The fit method calclulates the *statistics* necessary to later transform a dataset (i.e. perform actual imputatations). Inductive methods calculate statistic on the fit data, then impute new missing data with that value. All currently supported methods are inductive.

Parameters **X** (*pd.DataFrame*) – pandas DataFrame on which imputer is fit.

Returns instance of the PredictiveImputer class.

Return type self

fit_transform(*X*, *y=None*, ***trans_kwargs*)

Convenience method to fit then transform the same dataset.

n

Property getter to return the value of the n property.

transform(*X*, ***trans_kwargs*)

Impute each column within a DataFrame using fit imputation methods.

The transform step performs the actual imputations. Given a dataset previously fit, *transform* imputes each column with it's respective imputed values from fit (in the case of inductive) or performs new fit and transform in one sweep (in the case of transductive).

Parameters

- **X** (*pd.DataFrame*) – fit DataFrame to impute.
- ****trans_kwargs** – dict, optional args for bayesian.

Returns imputed in place or copy of original.

Return type X (pd.DataFrame)

Raises ValueError – same columns must appear in fit and transform.

4.4 Mice Imputer

class autoimpute.imputations.**MiceImputer**(*k=3*, *n=5*, *strategy='default predictive'*, *predictors='all'*, *imp_kwgs=None*, *seed=None*, *visit='default'*, *return_list=False*)

Techniques to impute Series with missing values multiple times using repeated fits and applications to reach a stable imputation.

The MiceImputer class implements multiple imputation, i.e., a series or repetition of applications of imputation to reach a stable imputation, similar to the functioning of the R package MICE. It leverages the methods found in the BaseImputer. This imputer passes all the work for each imputation to the SingleImputer, but it controls the arguments each imputer receives. The args are flexible depending on what the user specifies for each imputation.

Note that the Imputer allows for one imputation method per column only. Therefore, the behavior of *strategy* is the same as the SingleImputer, but the predictors are allowed to change for each imputation.

__init__(*k=3*, *n=5*, *strategy='default predictive'*, *predictors='all'*, *imp_kwgs=None*, *seed=None*, *visit='default'*, *return_list=False*)

Create an instance of the SeriesImputer class.

As with sklearn classes, all arguments take default values. Therefore, SeriesImputer() creates a valid class instance. The instance is used to set up an imputer and perform checks on arguments.

Parameters

- **k** (*int, optional*) – number of repeated fits and transformations to apply to reach a stable impution. Default is 3. Value must be greater than or equal to 1.
- **n** (*int, optional*) – number of imputations to perform. Default is 5. Value must be greater than or equal to 1.
- **strategy** (*str, iter, dict; optional*) – strategy for single imputer. Default value is str –> *predictive default*. See BaseImputer for all available strategies. If str, single strategy broadcast to all series in DataFrame. If iter, must provide 1 strategy per column. Each method w/in iterator applies to column with same index value in DataFrame. If dict, must provide key = column name, value = imputer. Dict the most flexible and PREFERRED way to create custom imputation strategies if not using the default. Dict does not require method for every column; just those specified as keys.
- **predictors** (*str, iter, dict, optional*) – defaults to all, i.e. use all predictors. If all, every column will be used for every class prediction. If a list, subset of columns used for all predictions. If a dict, specify which columns to use as predictors for

each imputation. Columns not specified in dict but present in *strategy* receive *all* other cols as preds.

- **imp_kwgs** (*dict, optional*) – keyword arguments for each imputer. Default is None, which means default imputer created to match specific strategy. imp_kwgs keys can be either columns or strategies. If strategies, each column given that strategy is instantiated with same arguments. When strategy is *default*, imp_kwgs is ignored.
- **seed** (*int, optional*) – seed setting for reproducible results. Defualt is None. No validation, but values should be integer.
- **return_list** (*bool, optional*) – return m as list or generator. Default is False. m imputations returned as generator. More memory efficient. return as list if return_list=True

k

Property getter to return the value of the k property.

transform(*X*)

Impute each column within a DataFrame using fit imputation methods.

The transform step performs the actual imputations. Given a dataset previously fit, *transform* imputes each column with it's respective imputed values from fit (in the case of inductive) or performs new fit and transform in one sweep (in the case of transductive). The transformations and fits are repeatedly applied and refitted self.k times to reach a stable imputation.

Parameters **X** (*pd.DataFrame*) – fit DataFrame to impute.

Returns imputed in place or copy of original.

Return type X (pd.DataFrame)

Raises ValueError – same columns must appear in fit and transform.

CHAPTER 5

Missingness Classifier

Module to predict missingness in data and generate imputation test cases.

This module contains the MissingnessClassifier, which is used to predict missingness within a dataset using information derived from other features. The MissingnessClassifier also generates test cases for imputation. Often, we do not and will never have the true value of a missing data point, so its challenging to validate an imputation model's performance. The MissingnessClassifer generates missing "test" samples from observed that have high likelihood of being missing, which a user can then "impute". This practice is useful to validate models that contain truly missing data.

class autoimpute.imputations.mis_classifier.**MissingnessClassifier**(*classifier=None*, *predictors='all'*)

Classify values as missing or not, based on missingness patterns.

The class has has numerous use cases. First, it fits columns of a DataFrame and predicts whether or not an observation is missing, based on all available information in other columns. The class supports both class prediction and class probabilities.

Second, the class can generate test cases for imputation analysis. Test cases are values that are truly observed but have a high probability of being missing. These cases make imputation process supervised as opposed to unsupervised. A user never knows the true value of missing data but can verify imputation methods on test cases for which the true value is known.

__init__(*classifier=None*, *predictors='all'*)

Create an instance of the MissingnessClassifier.

The MissingnessClassifier inherits from sklearn BaseEstimator and ClassifierMixin. This inheritence and this class' implementation ensure that the MissingnessClassifier is a valid classifier that will work in an sklearn pipeline.

Parameters

- **classifier** (*classifier, optional*) – valid classifier from sklearn. If None, default is xgboost. Note that classifier must conform to sklearn style. This means it must implement the *predict_proba* method and act as a porper classifier.

- **predictors** (*str, iter, dict, optiona*) – defaults to all, i.e. use all predictors. If all, every column will be used for every class prediction. If a list, subset of columns used for all predictions. If a dict, specify which columns to use as predictors for each imputation. Columns not specified in dict will receive *all* by default.

classifier
Property getter to return the value of the classifier property

fit(*X*, ***kwargs*)
Fit an individual classifier for each column in the DataFrame.

For each feature in the DataFrame, a classifier (default: xgboost) is fit with the feature as the response (y) and all other features as covariates (X). The resulting classifiers are stored in the class instance statistics. One *fit* for each column in the dataset. Column specification will be supported as well.

Parameters

- **X** (*pd.DataFrame*) – DataFrame on which to fit classifiers
- ****kwargs** – keyword arguments used by classifiers

Returns instance of MissingnessClassifier

Return type self

fit_predict(*X*)
Convenience method for fit and class prediction.

Parameters **X** (*pd.DataFrame*) – DataFrame to fit classifier and predict class.

Returns DataFrame of class predictions.

Return type pd.DataFrame

fit_predict_proba(*X*)
Convenience method for fit and class probability prediction.

Parameters **X** (*pd.DataFrame*) – DataFrame to fit classifier and prredict prob.

Returns DataFrame of class probability predictions.

Return type pd.DataFrame

gen_test_df(*X*, *thresh=0.5*, *m=0.05*, *inplace=False*, *use_exist=False*)
Generate new DatFrame with value of false positives set to missing.

Method generates new DataFrame with the locations (indices) of false positives set to missing. Utilizes *gen_test_indices* to detect index of false positives.

Parameters

- **X** (*pd.DataFrame*) – DataFrame from which test indices generated. Data first goes through *fit_predict_proba*.
- **thresh** (*float, optional*) – Threshhold for generating false positive. If raw value is observed and P(missing) >= thresh, then the observation is considered a false positive and index is stored.
- **m** (*float, optional*) – % false positive threshhold for warning. If % <= m, issue warning with % of test cases.
- **use_exist** (*bool, optional*) – Whether or not to use existing fit and classifiers. Default is False.

Returns DataFrame with false positives set to missing.

Return type pd.DataFrame

gen_test_indices (*X*, *thresh=0.5*, *use_exist=False*)
Generate indices of false positives for each fitted column.

Method generates the locations (indices) of false positives returned from classifiers. These are instances that have a high probability of being missing even though true value is observed. Use this method to get indices without mutating the actual DataFrame. To set the values to missing for the actual DataFrame, use *gen_test_df*.

Parameters

- **X** (*pd.DataFrame*) – DataFrame from which test indices generated. Data first goes through *fit_predict_proba*.
- **thresh** (*float, optional*) – Threshhold for generating false positive. If raw value is observed and P(missing) >= thresh, then the observation is considered a false positive and index is stored.
- **use_exist** (*bool, optional*) – Whether or not to use existing fit and classifiers. Default is False.

Returns test_indice available from *self.test_indices*

Return type self

predict (*X*, ***kwargs*)
Predict class of each feature. 1 for missing; 0 for not missing.

First checks to ensure data has been fit. If fit, *predict* method uses the respective classifier of each feature (stored in statistics) and predicts class membership for each observation of each feature. 1 = missing; 0 = not missing. Prediction is binary, as class membership is hard. If probability deesired, use *predict_proba* method.

Parameters

- **X** (*pd.DataFrame*) – DataFrame used to create predictions.
- **kwargs** – kewword arguments. Used by the classifer.

Returns DataFrame with class prediction for each observation.

Return type pd.DataFrame

predict_proba (*X*, ***kwargs*)
Predict probability of missing class membership of each feature.

First checks to ensure data has been fit. If fit, *predict_proba* method uses the respsective classifier of each feature (in statistics) and predicts probability of missing class membership for each observation of each feature. Prediction is probability of missing. Therefore, probability of not missing is 1-P(missing). For hard class membership prediction, use *predict*.

Parameters **X** (*pd.DataFrame*) – DataFrame used to create probabilities.

Returns

DataFrame with probability of missing class for each observation.

Return type pd.DataFrame

CHAPTER 6

Analysis Models

This section documents analysis models within `Autoimpute` and their respective diagnostics.

The `MiLinearRegression` and `MiLogisticRegression` extend linear and logistic regression to multiply imputed datasets. Under the hood, each regression class uses a `MiceImputer` to handle missing data prior to supervised analysis. Users of each regression class can tweak the underlying `MiceImputer` through the `mi_kwgs` argument or pass a pre-configured instance to the `mi` argument (recommended).

Users can also specify whether the classes should use `sklearn` or `statsmodels` to implement linear or logistic regression. The default is `statsmodels`. When used, end users get more detailed parameter diagnostics for regression on multiply imputed data.

Finally, this section provides diagnostic helper methods to assess bias of parameters from a regression model.

6.1 Linear Regression for Multiply Imputed Data

class autoimpute.analysis.**MiLinearRegression**(*mi=None*, *model_lib='statsmodels'*, *mi_kwgs=None*, *model_kwgs=None*)

Linear Regression wrapper for multiply imputed datasets.

The MiLinearRegression class wraps the sklearn and statsmodels libraries to extend linear regression to multiply imputed datasets. The class wraps statsmodels as well as sklearn because sklearn alone does not provide sufficient functionality to pool estimates under Rubin's rules. sklearn is for machine learning; therefore, important inference capabilities are lacking, such as easily calculating std. error estimates for parameters. If users want inference from regression analysis of multiply imputed data, utilze the statsmodels implementation in this class instead.

linear_models
linear models used by supported python libs.

Type dict

__init__(*mi=None*, *model_lib='statsmodels'*, *mi_kwgs=None*, *model_kwgs=None*)
Create an instance of the Autoimpute MiLinearRegression class.

Parameters

- **mi** (*MiceImputer, Optional*) – An instance of a MiceImputer. Default is none. Can create one through *mi_kwgs* instead.
- **model_lib** (*str, Optional*) – library the regressor will use to implement regression. Options are sklearn and statsmodels. Default is statsmodels.
- **mi_kwgs** (*dict, Optional*) – keyword args to instantiate MiceImputer. Default is None. If valid MiceImputer passed as mi argument, then mi_kwgs ignored.
- **model_kwgs** (*dict, Optional*) – keyword args to instantiate regressor. Default is None.

Returns self. Instance of the class.

fit(*X*, *y*)

Fit model specified to multiply imputed dataset.

Fit a linear regression on multiply imputed datasets. The method first creates multiply imputed data using the MiceImputer instantiated when creating an instance of the class. It then runs a linear model on each m datasets. The linear model comes from sklearn or statsmodels. Finally, the fit method calculates pooled parameters from the m linear models. Note that variance for pooled parameters using Rubin's rules is available for statsmodels only. sklearn does not implement parameter inference out of the box. Autoimpute sklearn pooling TBD.

Parameters

- **X** (*pd.DataFrame*) – predictors to use. can contain missingness.
- **y** (*pd.Series, pd.DataFrame*) – response. can contain missingness.

Returns self. Instance of the class

predict(*X*)

Make predictions using statistics generated from fit.

The regression uses the pooled parameters from each of the imputed datasets to generate a set of single predictions. The pooled params come from multiply imputed datasets, but the predictions themselves follow the same rules as an ordinary linear regression.

Parameters **X** (*pd.DataFrame*) – data to make predictions using pooled params.

Returns predictions.

Return type np.array

summary()

Provide a summary for model parameters, variance, and metrics.

The summary method brings together the statistics generated from fit as well as the variance ratios, if available. The statistics are far more valuable when using statsmodels than sklearn.

Returns summary statistics

Return type pd.DataFrame

6.2 Logistic Regression for Multiply Imputed Data

class autoimpute.analysis.**MiLogisticRegression**(*mi=None*, *model_lib='statsmodels'*, *mi_kwgs=None*, *model_kwgs=None*)

Logistic Regression wrapper for multiply imputed datasets.

The MiLogisticRegression class wraps the sklearn and statsmodels libraries to extend logistic regression to multiply imputed datasets. The class wraps statsmodels as well as sklearn because sklearn alone does not provide sufficient functionality to pool estimates under Rubin's rules. sklearn is for machine learning; therefore, important inference capabilities are lacking, such as easily calculating std. error estimates for parameters. If users want inference from regression analysis of multiply imputed data, utilze the statsmodels implementation in this class instead.

logistic_models
logistic models used by supported python libs.

Type dict

__init__ (*mi=None*, *model_lib='statsmodels'*, *mi_kwgs=None*, *model_kwgs=None*)
Create an instance of the Autoimpute MiLogisticRegression class.

Parameters

- **mi** (*MiceImputer, Optional*) – An instance of a MiceImputer. Default is None. Can create one through *mi_kwgs* instead.
- **model_lib** (*str, Optional*) – library the regressor will use to implement regression. Options are sklearn and statsmodels. Default is statsmodels.
- **mi_kwgs** (*dict, Optional*) – keyword args to instantiate MiceImputer. Default is None. If valid MiceImputer passed as *mi* argument, then *mi_kwgs* ignored.
- **model_kwgs** (*dict, Optional*) – keyword args to instantiate regressor. Default is None.

Returns self. Instance of the class.

fit (*X*, *y*)
Fit model specified to multiply imputed dataset.

Fit a logistic regression on multiply imputed datasets. The method creates multiply imputed data using the MiceImputer instantiated when creating an instance of the class. It then runs a logistic model on m datasets. The logistic model comes from sklearn or statsmodels. Finally, the fit method calculates pooled parameters from m logistic models. Note that variance for pooled parameters using Rubin's rules is available for statsmodels only. sklearn does not implement parameter inference out of the box.

Parameters

- **X** (*pd.DataFrame*) – predictors to use. can contain missingness.
- **y** (*pd.Series, pd.DataFrame*) – response. can contain missingness.

Returns self. Instance of the class

predict (*X*, *threshold=0.5*)
Make predictions using statistics generated from fit.

The predict method calls on the predict_proba method, which returns the probability of class membership for each prediction. These probabilities range from 0 to 1. Therefore, anything below the set threshold is assigned to class 0, while anything above the threshold is assigned to class 1. The deafult threshhold is 0.5, which indicates a balanced dataset.

Parameters

- **X** (*pd.DataFrame*) – data to make predictions using pooled params.
- **threshold** (*float, Optional*) – boundary for class membership. Default is 0.5. Values can range from 0 to 1.

Returns predictions.

Return type np.array

predict_proba(*X*)

Predict probabilities of class membership for logistic regression.

The regression uses the pooled parameters from each of the imputed datasets to generate a set of single predictions. The pooled params come from multiply imputed datasets, but the predictions themselves follow the same rules as an logistic regression. Because this is logistic regression, the sigmoid function is applied to the result of the normal equation, giving us probabilities between 0 and 1 for each prediction. This method returns those probabilities.

Parameters **X** (*pd.Dataframe*) – predictors to predict response

Returns prob of class membership for predicted observations.

Return type np.array

summary()

Provide a summary for model parameters, variance, and metrics.

The summary method brings together the statistics generated from fit as well as the variance ratios, if available. The statistics are far more valuable when using statsmodels than sklearn.

Returns summary statistics

Return type pd.DataFrame

6.3 Diagnostics

autoimpute.analysis.**raw_bias**(*Q_bar*, *Q*)

Calculate raw bias between coefficients Q and actual Q.

Q_bar can be one estimate (scalar) or a vector of estimates. This equation subtracts the expected Q_bar from Q, element-wise. The result is the bias of each coefficient from its true value.

Parameters

- **Q_bar** (*number, array*) – single estimate or array of estimates.
- **Q** (*number, array*) – single truth or array of truths.

Returns element-wise difference between estimates and truths.

Return type scalar, array

Raises

- ValueError – Shape mismatch
- ValueError – Q_bar and Q not the same length

autoimpute.analysis.**percent_bias**(*Q_bar*, *Q*)

Calculate precent bias between coefficients Q and actual Q.

Q_bar can be one estimate (scalar) or a vector of estimates. This equation subtracts the expected Q_bar from Q, element-wise. The result is the bias of each coefficient from its true value. We then divide this number by Q itself, again in element-wise fashion, to produce % bias.

Parameters

- **Q_bar** (*number, array*) – single estimate or array of estimates.
- **Q** (*number, array*) – single truth or array of truths.

Returns element-wise difference between estimates and truths.

Return type scalar, array

Raises

- `ValueError` – Shape mismatch
- `ValueError` – Q_bar and Q not the same length

CHAPTER 7

Visualization Methods

This section documents visualization methods within `Autoimpute`.

Visualization methods support all functionality within `Autoimpute`, from missing data exploration to imputation analysis. The documentation below breaks down each visualization method and groups them into their respsective categories. The categories represent other modules within `Autoimpute`.

NOTE: The visualization module is currently under development. While the functions outlined below are stable in `0.12.x`, they might change thereafterr.

7.1 Utility

`Autoimpute` comes with a number of *utility methods* to examine missing data before imputation takes place. This package supports these methods with a number of visualization techniques to explore patterns within missing data. The primary techniques wrap the excellent missingno package. `Autoimpute` simply leverages `missingno` to make its offerings familiar in this packages' API design. The methods appear below:

`autoimpute.visuals.`**`plot_md_locations`**(*data*, ***kwargs*)

Plot the locations where data is missing within a DataFrame.

Parameters

- **`data`** (*`pd.DataFrame`*) – DataFrame to plot.
- **`**kwargs`** – Keyword arguments for plot. Passed to missingno.matrix.

Returns missingness location plot.

Return type matplotlib.axes._subplots.AxesSubplot

Raises `TypeError` – if data is not a DataFrame. Error raised through decorator.

`autoimpute.visuals.`**`plot_md_percent`**(*data*, ***kwargs*)

Plot the percentage of missing data by column within a DataFrame.

Parameters

- **data** (*pd.DataFrame*) – DataFrame to plot.
- ****kwargs** – Keyword arguments for plot. Passed to missingno.bar.

Returns missingness percent plot.

Return type matplotlib.axes._subplots.AxesSubplot

Raises TypeError – if data is not a DataFrame. Error raised through decorator.

autoimpute.visuals.**plot_nullility_corr**(*data*, ***kwargs*)

Plot the nullility correlation of missing data within a DataFrame.

Parameters

- **data** (*pd.DataFrame*) – DataFrame to plot.
- ****kwargs** – Keyword arguments for plot. Passed to missingno.heatmap.

Returns nullility correlation plot.

Return type matplotlib.axes._subplots.AxesSubplot

Raises

- TypeError – if data is not a DataFrame. Error raised through decorator.
- ValueError – dataset fully observed. Raised through helper method.

autoimpute.visuals.**plot_nullility_dendogram**(*data*, ***kwargs*)

Plot the nullility dendogram of missing data within a DataFrame.

Parameters

- **data** (*pd.DataFrame*) – DataFrame to plot.
- ****kwargs** – Keyword arguments for plot. Passed to missingno.dendogram.

Returns nullility dendogram plot.

Return type matplotlib.axes._subplots.AxesSubplot

Raises

- TypeError – if data is not a DataFrame. Error raised through decorator.
- ValueError – dataset fully observed. Raised through helper method.

7.2 Imputation

Two main classes within Autoimpute are the *SingleImputer and MultipleImputer*. The visualization module within this package contains a number of techniques to visually assess the quality and performance of these imputers. The important methods appear below:

autoimpute.visuals.helpers.**_validate_data**(*d*, *mi*, *imp_col=None*)

Private helper method to validate data vs multiple imputations.

Parameters

- **d** (*list*) – dataset returned from multiple imputation.
- **mi** (MultipleImputer) – multiple imputer used to generate d.
- **imp_col** (*str*) – column to plot. Should be a column with imputations.

Raises

- ValueError – d should be list of tuples returned from mi transform.
- ValueError – mi should be instance of MultipleImputer used to produce d.
- ValueError – mi should have **imputed_** attribute after transformation.
- ValueError – Number of imputations should equal length of the dataset.
- ValueError – Columns in each imputed data should be the same.
- ValueError – Colums in each imputed data should be same as **mi.imputed_**.
- ValueError – imp_col must be in both datasets and **mi.imputed_** keys.

autoimpute.visuals.**plot_imp_scatter**(*d*, *x*, *y*, *strategy*, *color=None*, *title='Jointplot after Imputation'*, *h=8.27*, *imp_kwgs=None*, *a=0.5*, *marginals=None*, *obs_color='navy'*, *imp_color='red'*, ***plot_kwgs*)

Plot the joint scatter and density plot after single imputation.

Use this method to visualize a scatterplot between two features, x and y, where y is imputed and x is a predictor used to impute y. This method performs single imputation and is useful to determine how an imputation method looks under the hood.

Parameters

- **d** (*pd.DataFrame*) – DataFrame with data to impute and plot.
- **x** (*str*) – column to plot on x axis.
- **y** (*str*) – column to plot on y axis and set color for imputation.
- **strategy** (*str*) – imputation method for SingleImputer.
- **color** (*str, Optional*) – which variable to color with imputations. Deafult is none, which means y is colored. Other option is to color "x". Color should be the same as "x" or "y".
- **title** (*str, Optional*) – title of plot. "Defualt is Jointplot after Imputation".
- **h** (*float, Optional*) – height of the jointplot. Default is 8.27
- **imp_kwgs** (*dict, Optional*) – imp kwgs for SingleImputer procedure. Default is None.
- **a** (*float, Optional*) – alpha for plot color. Default is 0.5
- **marginals** (*dict, Optional*) – dictionary of marginal plot args. Default is None, configured in code below.
- **obs_color** (*str, Optional*) – color of observed. Default is navy.
- **imp_color** (*str, Optional*) – color of imputations. Default is red.
- ****plot_kwgs** – keyword arguments used by sns.set.

Raises ValueError – x and y must be names of columns in data

autoimpute.visuals.**plot_imp_dists**(*d*, *mi*, *imp_col*, *title='Distributions after Imputation'*, *include_observed=True*, *separate_observed=True*, *side_by_side=False*, *hist_observed=False*, *hist_imputed=False*, *gw=(0.5, 0.5)*, *gh=(0.5, 0.5)*, ***plot_kwgs*)

Plot the density between imputations for a given column.

Use this method to plot the density of a given column after multiple imputation. The function allows the user to also plot the observed data from the column prior to imputation taking place. Further, the user can specify whether the observed should be separated into its own plot or not.

Parameters

- **d** (*list*) – dataset returned from multiple imputation.
- **mi** (MultipleImputer) – multiple imputer used to generate d.
- **imp_col** (*str*) – column to plot. Should be a column with imputations.
- **title** (*str, Optional*) – title of plot. Default is "Distributions after Imputation".
- **include_observed** (*bool, Optional*) – whether or not to include observed data in the plot. Default is True. If False, observed data for imp_col will not be included as a distribution for density.
- **separate_observed** (*bool, Optional*) – whether or not to separate the observed data when plotting against imputed. Default is True. If False, observed data distribution will be plotted on same plot as the imputed data distribution. Note, this attribute matters if and only if *include_observed=True*.
- **side_by_side** (*bool, Optional*) – whether columns should be plotted next to each other or stacked vertically. Default is False. If True, plots will be plotted side-by-side. Note, this attribute matters if and only if *include_observed=True*.
- **hist_observed** (*bool, Optional*) – whether histogram should be plotted along with the density for observed values. Default is False. Note, this attribute matters if and only if *include_observed=True*.
- **hist_imputed** (*bool, Optional*) – whether histogram should be plotted along with the density for imputed values. Default is False. Note, this attribute matters if and only if *include_observed=True*.
- **gw** (*tuple, Optional*) – if side-by-side plot, the width ratios for each plot. Default is (.5, .5), so each plot will be same width. Matters if and only if *include_observed=True* and *side_by_side=True*.
- **gh** (*tuple, Optional*) – if stacked plot, the height ratios for each plot. Default is (.5, .5), so each plot will be the same height. Matters if and only if *include_observed=True* and *side_by_side=False*.
- ****plot_kwgs** – keyword arguments used by sns.set.

Returns densityplot for observed and/or imputed data

Return type sns.distplot

Raises ValueError – see _validate_data method

autoimpute.visuals.**plot_imp_boxplots**(*d*, *mi*, *imp_col*, *side_by_side=False*, *title='Observed vs. Imputed Boxplots'*, *obs_kwgs=None*, *imp_kwgs=None*, ***plot_kwgs*)

Plot the boxplots between observed and imputations for a given column.

Use this method to plot the boxplots of a given column after multiple imputation. The function also plots the boxplot of the observed data from the column prior to imputation taking place. Further, the user can specify additional arguments to tailor the design of the plots themselves.

Parameters

- **d** (*list*) – dataset returned from multiple imputation.
- **mi** (MultipleImputer) – multiple imputer used to generate d.

- **imp_col** (*str*) – column to plot. Should be a column with imputations.
- **side_by_side** (*bool, Optional*) – whether columns should be plotted next to each other or stacked vertically. Default is False. If True, plots will be plotted side-by-side.
- **title** (*str, Optional*) – title of boxplots. Default is "Observed vs. Imputed Boxplots."
- **obs_kwgs** (*dict, Optional*) – dictionary of arguments to unpack for observed boxplot. Default is None, so no additional tailoring.
- **imp_kwgs** (*dict, Optional*) – dictionary of arguments to unpack for imputed boxplots. Default is None, so no additional tailoring.
- ****plot_kwgs** – keyword arguments used by sns.set.

Returns boxplots for observed and imputed data

Return type sns.distplot

Raises ValueError – see _validate_data method.

autoimpute.visuals.**plot_imp_swarm**(*d*, *mi*, *imp_col*, *palette=None*, *title='Imputation Swarm'*, ***plot_kwgs*)

Create the swarm plot for multiply imputed data.

Parameters

- **d** (*list*) – dataset returned from multiple imputation.
- **mi** (MultipleImputer) – multiple imputer used to generate d.
- **imp_col** (*str*) – column to plot. Should be a column with imputations.
- **title** (*str, Optional*) – title of plot. Default is "Imputation Swarm".
- **palette** (*list, tuple, Optional*) – colors for the imps and observed. Default is None. if None, colors default to ["r","c"].
- ****plot_kwgs** – keyword arguments used by sns.set.

Returns swarmplot for imputed data

Return type sns.distplot

Raises ValueError – see _validate_data method.

autoimpute.visuals.**plot_imp_strip**(*d*, *mi*, *imp_col*, *palette=None*, *title='Imputation Strip'*, ***plot_kwgs*)

Create the strip plot for multiply imputed data.

Parameters

- **d** (*list*) – dataset returned from multiple imputation.
- **mi** (MultipleImputer) – multiple imputer used to generate d.
- **imp_col** (*str*) – column to plot. Should be a column with imputations.
- **title** (*str, Optional*) – title of plot. Default is "Imputation Strip".
- **palette** (*list, tuple, Optional*) – colors for the imps and observed. Default is None. if None, colors default to ["r","c"].
- ****plot_kwgs** – keyword arguments used by sns.set.

Returns stripplot for imputed data

Return type sns.distplot

Raises `ValueError` – see _validate_data method.

We are actively developing *Autoimpute*, so sometimes the docs fall behind. The *README* is always up to date, as is the master branch. Therefore, consult those first. If you find discrepancies between the docs and the package, please still let us know!

Python Module Index

a

Index

Symbols

G

I

K

L

M

N

O

P

R

S

T

U

V